Universidad
Andrés Bello

ARREGLOS (VECTORES y MATRICES) DINÁMICOS

CADENAS DINÁMICAS (caso particular de arreglo)

Ingeniería en Computación e Informática

Arreglos Dinámicos

- Para definir durante la ejecución del programa, arreglos cuyo tamaño es exactamente el que el usuario necesita.

- Se Utilizan para ello dos funciones de la biblioteca estándar (`stdlib.h`):
  - `malloc()` : Abreviatura de *memory allocate*, que podemos traducir por “reservar memoria”. Ésta solicita un bloque de memoria del tamaño que se indique (en *bytes*).
  - `free()` : Que en inglés significa “liberar”. libera memoria obtenida con malloc, es decir, la marca como disponible para futuras llamadas a `malloc`.

Arreglos Dinámicos

Ejemplo

```c
#include <stdlib.h>
#include <stdio.h>
int main( int argc , char *argv []) {
  int *a; //puntero a varios enteros (porque se inicializa con malloc más adelante)
  int n, i ;
  printf ( "Número de elementos: " ) ;
  scanf ( " %d" , &n) ; //lee n (el tamaño del arreglo)

  //Al puntero a se le asigna memoria dinámica (n tamaños enteros)
  //Por lo tanto la variable a es un arreglo (vector) dinámico de enteros
  a = malloc(n * sizeof ( int ) ) ;

  //asigna valores a cada posición del arreglo
  for ( i =0; i<n; i ++) {
    *(a+i) = i ; // también puede ser: a[i] = i
  }
  //imprime los elementos(valores) asignados al vector
  for ( i =0; i<n; i ++) {
    printf("\nValor asignado a la posición %d del arreglo:  %d", i, *(a+i));   //a[i]
  }

 free (a) ; //Se libera la memoria
```

Arreglos Dinámicos

Explicación

- `int *a;`
- No se trata de un puntero a un entero, sino de un puntero a una secuencia de enteros.
- Ambos conceptos son equivalentes en C, pues ambos son meras direcciones de memoria.
- La variable `a` es un arreglo dinámico de enteros, pues su memoria se obtiene dinámicamente.
- Esto es, en tiempo de ejecución y según convenga a las necesidades.
- No sabemos aún cuántos enteros serán apuntados por `a`, ya que el tamaño no se conocerá hasta que se ejecute el programa y se lea por teclado.

Arreglos Dinámicos

Explicación

- La función `malloc` espera recibir como argumento un número entero: el número de *bytes* que queremos reservar.
- Si deseamos reservar `n` valores de tipo `int`, hemos de solicitar memoria para `n * sizeof (int)` *bytes*.
- Recordemos que `sizeof(int)` es la ocupación en *bytes* de un tipo de dato (en caso de `int` sabemos que es de 4).

Arreglos Dinámicos

Explicación

- La función `free` recibe un puntero a cualquier tipo de datos: la dirección de memoria en la que empieza un bloque previamente obtenido con una llamada a `malloc`.
- Lo que hace `free` es liberar ese bloque de memoria, es decir, considerar que pasa a estar disponible para otras posibles llamadas a `malloc`.
- Es como cerrar un archivo: si no necesito un recurso, lo libero para que otros lo puedan aprovechar.
- Podemos así aprovechar la memoria de forma óptima.
- **Importante: el programa debe efectuar una llamada a `free` por cada llamada a `malloc`.**

- Si vas a seguir ocupando el puntero, conviene que después de hacer `free` asignes al puntero el valor `NULL`.

- `free` libera la memoria apuntada por un puntero, pero no modifica el valor de la variable que se le pasa.

- Imaginemos que un bloque de memoria de 10 enteros que empieza en la dirección 1000 es apuntado por una variable `a` de tipo `int *`, es decir, imagina que vale 1000.

- Cuando ejecutamos `free(a)`, ese bloque se libera y pasa a estar disponible para eventuales llamadas a `malloc`.

Arreglos Dinámicos

`Free`

- Pero ¡`a` sigue valiendo 1000! ¿Por qué? Porque a se ha pasado a `free` **por valor, no por referencia**, así que `free` no tiene forma de modificar el valor de `a`.
- Es recomendable que asignes a `a` el valor `NULL` después de una llamada a `free`, eso hace explícito que la variable `a` no apunta a nada.
- **Recuerden, es responsabilidad de uno y conviene hacerlo: asignar explícitamente el valor `NULL` a todo puntero que no apunte a memoria reservada.**

- La función `malloc` puede fallar por diferentes motivos.
- Podemos saber cuándo ha fallado nos devuelve el valor NULL.
- Imaginemos que solicitas 2 *megabytes* de memoria en un computador que sólo dispone de 1 *megabyte*.
- En tal caso, la función `malloc` devolverá el valor `NULL` para indicar que no pudo efectuar la reserva de memoria solicitada.

Arreglos Dinámicos

malloc

- Los programas correctamente escritos deben comprobar si se pudo obtener la memoria solicitada y, en caso contrario, tratar el error.

```
1 a = malloc(n * sizeof (int));
2 if(a == NULL)
3   printf ("Error: no hay memoria suficiente\n");
4 else {
5   ...
6 }
```

- Es posible solicitar la memoria y comprobar si se pudo obtener en una única línea:

```
1 if((a = malloc(talla * sizeof (int))) == NULL)
2   printf ("Error: no hay memoria suficiente\n");
3
4 else {
5   ...
6 }
```

Arreglos Dinámicos

`calloc`

- La función `calloc` es similar a `malloc`, pero presenta un prototipo diferente y hace algo más que reservar memoria: la inicializa a cero.
- Se define:

```
1  void * calloc(int nmemb, int size);
```

- Con `calloc`, puedes pedir memoria para un arreglo de `n` enteros así:

```
1  a = calloc(n, sizeof (int));
```

- El primer parámetro es el número de elementos y el segundo, el número de *bytes* que ocupa cada elemento.
- No hay que multiplicar una cantidad por otra, como en el `malloc`.
- Todos los enteros del arreglo se inicializan a cero.

Arreglos Dinámicos

calloc

- Es como si ejecutáramos este fragmento de código:

```
1 a = malloc(n * sizeof (int));
2 for(i = 0; i < n; i++)
3   a[i] = 0;
```

- ¿Por qué no usar siempre `calloc`, si parece mejor que `malloc`?

- Por eficiencia. En ocasiones no desearías que se pierda tiempo de ejecución inicializando la memoria a cero, ya que uno mismo necesitará inicializarla a otros valores inmediatamente.

- Recuerda que garantizar la mayor eficiencia de los programas es uno de los objetivos del lenguaje de programación C.

Ejercicios

1. Desarrolle una función que reciba un entero n y devuelva un arreglo de largo n con los primeros n números de Fibonacci

$$f_n = \begin{cases} 0 & si \quad n = 0 \\ 1 & si \quad n = 1 \\ f_{n-1} + f_{n-2} & si \quad n > 1 \end{cases}$$

2. Desarrolle una función que reciba un arreglo dinámico y recorra sus elementos verificando la paridad de sus elementos.

Cadenas dinámicas

- Las cadenas son un caso particular de arreglos.
- Podemos usar cadenas de cualquier longitud gracias a la gestión de memoria dinámica.
- El siguiente programa, lee dos cadenas y construye una nueva que resulta de concatenar a éstas.

Cadenas dinámicas

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define CAPACIDAD 80

int main( int argc , char *argv []) {
   char cadena1[CAPACIDAD+1]; //string de largo fijo (no dinámico)
   char cadena2[CAPACIDAD+1]; //otro string de largo fijo (no dinámico)
   char *cadena3; //puntero que apunta a una secuencia de caracteres (arreglo dinámico)

   printf ("Ingrese string 1: " );
   fflush(stdin); //vacíamos el buffer de entrada
   scanf("%s",&cadena1); //se usa "%[^\n]s" para string con espacios;

   printf ("Ingrese string 2: " );
   fflush(stdin); //vacíamos el buffer de entrada
   scanf("%s",&cadena2);

   //tamaño de cadena3 (como es una arreglo dinámico) depende del largo (strlen) de cadena1 y cadena2
   cadena3 = malloc (( strlen (cadena1) + strlen (cadena2) + 1) * sizeof (char) ) ;

   strcpy (cadena3, cadena1) ; //copia los elementos de cadena1 a cadena3
   strcat (cadena3, cadena2) ; //concatena los elementos de cadena3 con cadena2

   printf ("Resultado de concatenar ambos: %s\n", cadena3) ;
   free (cadena3) ;
   return 0;
}
```

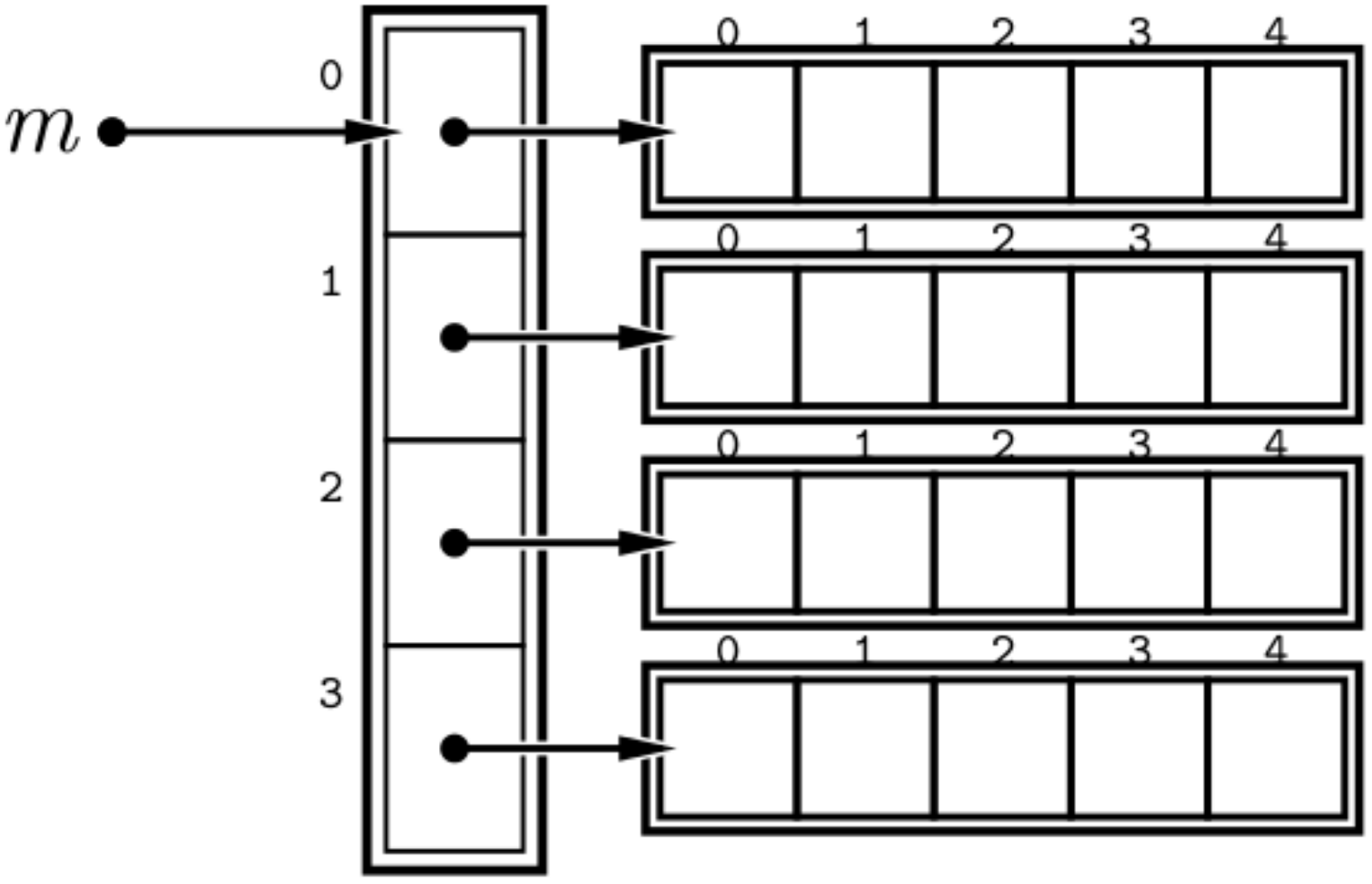
Matrices dinámicas

- Podemos extender la idea de los arreglos dinámicos a matrices dinámicas.
- Pero el asunto se complica notablemente: no podemos gestionar la matriz como una sucesión de elementos contiguos, sino como un “arreglo dinámico de arreglos dinámicos”.
- Analicemos el siguiente código.

Matrices dinámicas

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  float ** mat; /*matriz*/
  int i, j, n /*filas*/ , m/*columnas*/;
  printf ( " Filas : " ) ;
  scanf( " %d" , &n ) ; //Lee la cantidad de filas de la matriz
  printf ( " Columnas : " ) ;
  scanf( " %d" , &m ) ; //Lee la cantidad de columnas de la matriz

  /* reserva memoria para las filas de la matriz*/
  mat = malloc( n * sizeof ( float *)) ;

  for (i=0; i<n ; i ++) {
    //Por cada fila se reserva memoria para las columnas
    mat[i] = malloc(m * sizeof ( float ) ) ; //Cada mat[i] es un puntero
  }
  /* Asignar valores a la matriz*/
  for (i=0; i<n ; i++) { //Recorre cada fila i
     for (j=0; j<m ; j++) { //Para cada fila i recorre sus columnas j
       printf("\nIngrese valor columna %d de la fila %d: ", j, i);
       scanf("%f", &mat[i][j]);
     }
  }
  /*Mostrar los valores de las columnas de cada fila*/
  for (i=0; i<n ; i++) { //Recorre cada fila i
     for (j=0; j<m ; j++) //Para cada fila i recorre sus columnas j
       printf("\nValor columna %d de la fila %d:  %2.1f", j, i, mat[i][j]);
  }
  /* liberación de memoria */
  for (i=0; i<n ; i ++)
    free (mat[i]) ; //libera la memoria de cada fila
  free (mat) ; //libera la memoria de la matriz
  return 0;
}
```

Matrices dinámicas



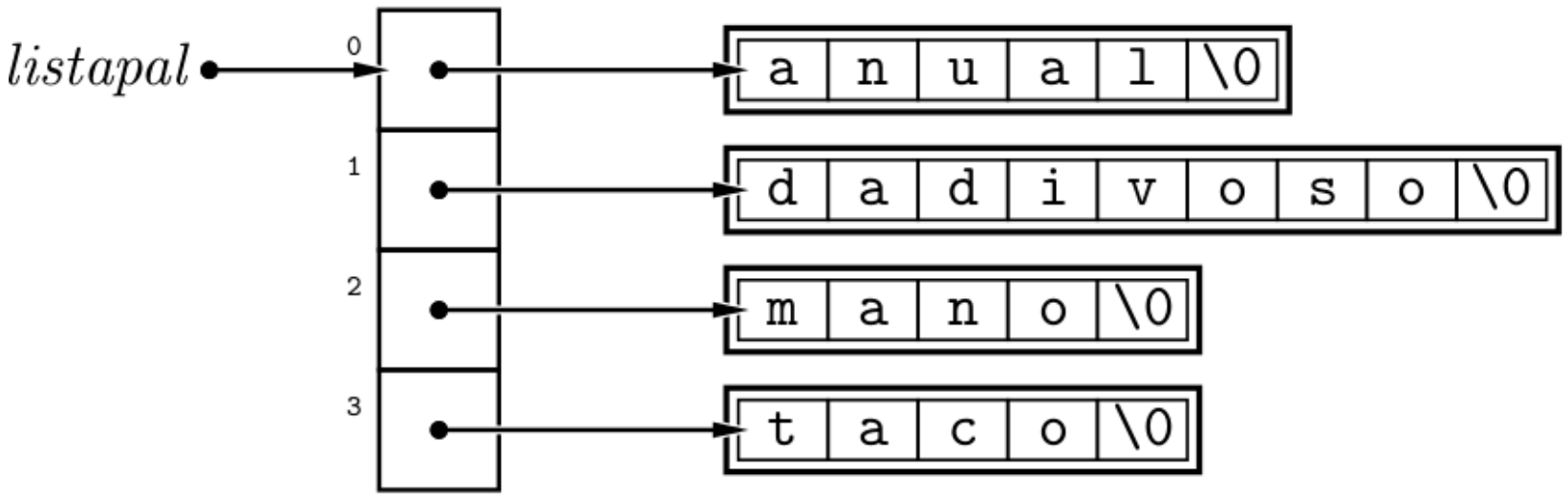
Matrices dinámicas

Arreglos de arreglos de tamaños arbitrarios

- Hemos aprendido a definir matrices dinámicas con un arreglo dinámico de arreglos dinámicos.
- El primero contiene punteros que apuntan a cada columna.
- Una característica de las matrices es que todas las filas tienen el mismo número de elementos (el número de columnas).
- Hay estructuras similares a las matrices pero que no imponen esa restricción.
- Pensemos, por ejemplo, en una lista de palabras.
- Una forma de almacenarla en memoria es como se ve en la siguiente figura.

Matrices dinámicas

Arreglos de arreglos de tamaños arbitrarios



Matrices dinámicas

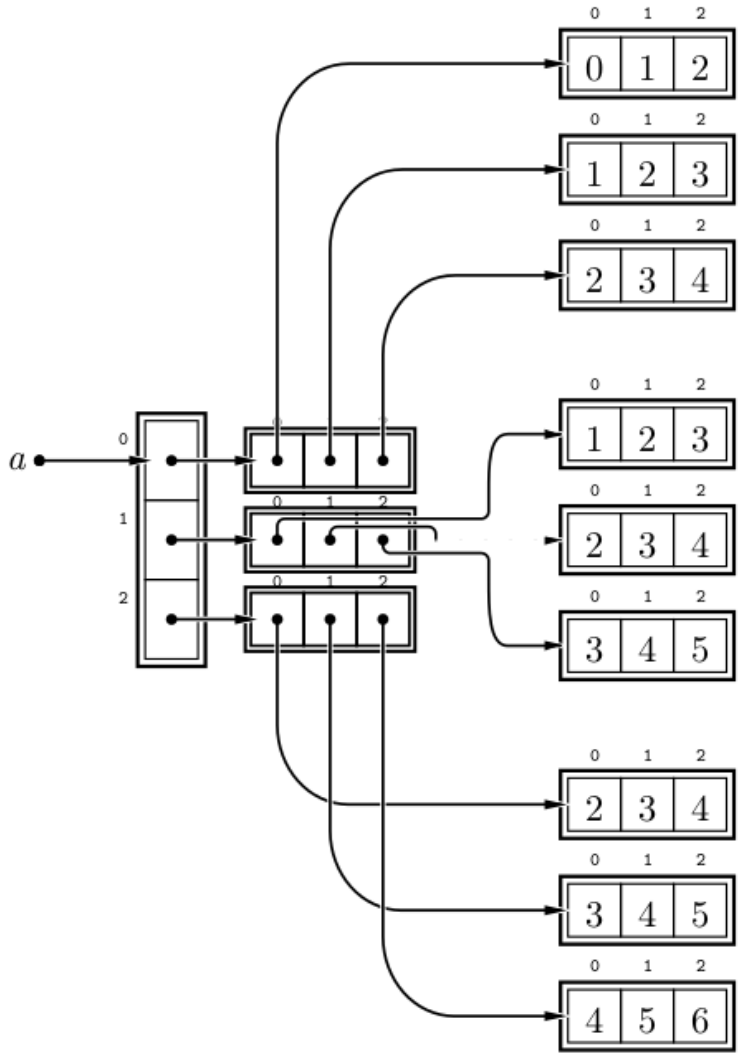
Arreglos de arreglos de tamaños arbitrarios

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define PALS 4
int main( int argc , char *argv []) {
  char **listapal ; //matriz de la lista de palabras (4 palabras)
  char linea [100]; //Largo del string fijo (no dinámico)
  int i ;

  /* Pedir memoria y leer datos */
  listapal = malloc(PALS * sizeof (char *)) ; //reserva espacio para 4 palabras

  for ( i =0; i<PALS; i ++)  {
    printf ("Teclea una palabra :" );
    scanf( " %s" , linea ) ;
    //Reserva memoria dinámica para la palabra que acaba de leer
    //Es decir: el tamaño de cada palabra es distinto
    listapal [i] = malloc (( strlen ( linea )+1) * sizeof (char) ) ;
    strcpy (listapal[i] , linea ) ; //asigna palabra al string de la fila i
  }
  /* Mostrar el contenido de la lista */
  for ( i =0; i<PALS; i ++)
    printf ( "Palabra % i : %s\n ", i , listapal [i]) ;

  /* Liberar memoria */
  for ( i =0; i<PALS; i ++)
    free ( listapal [ i ]) ;
  free ( listapal ) ;

  return 0;
}
```

Arreglos n-dimensionales

- Hemos considerado la creación de estructuras bidimensionales (matrices o arreglos de arreglos).
- Pero nada impide definir estructuras con más dimensiones.
- Acá un ejemplo sencillo donde se ilustra la idea creando una estructura dinámica con $3 \times 3 \times 3$ elementos, inicializarla, mostrar a su contenido y liberar la memoria ocupada.

Arreglos n-dimensionales



Formar

Transformar

Universidad Andrés Bello

Arreglos n-dimensionales

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  /* Tres asteriscos: arreglo de
     arreglos de arreglos de enteros. */
  int ***a;
  int i, j, k;

  /* Reserva de memoria */
  a = malloc(3*sizeof (int **));
  for (i=0; i<3; i++)
    {
      a[i] = malloc(3*sizeof (int *));
      for (j=0; j<3; j++)
        a[i][j] = malloc(3*sizeof (int));
    }

  /* Inicialización */
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      for (k=0; k<3; k++)
        a[i][j][k] = i+j+k;
```

```c
  /* Impresión */
  for(i=0; i<3; i++)
    for(j=0; j<3; j++)
      for(k=0; k<3; k++)
        printf ("%d %d %d: %d\n",
                i, j, k, a[i][j][k]);

  /* Liberación de memoria. */
  for(i=0; i<3; i++)
    {
      for(j=0; j<3; j++)
        free(a[i][j]);
      free(a[i]);
    }
```

Redimensionar la reserva de memoria

- Muchos programas no pueden determinar el tamaño de sus arreglos antes de empezar a trabajar con ellos.
- Por ejemplo, cuando se inicia la ejecución de un programa que gestiona una agenda telefónica no sabemos cuántas entradas contendrá finalmente.
- Podemos fijar un número máximo de entradas y pedir memoria para ellas con `malloc`.
- Pero entonces estaremos reproduciendo el problema que nos llevó a presentar los arreglos dinámicos.
- Afortunadamente, C permite que el tamaño de un arreglo cuya memoria se ha solicitado previamente con `malloc` crezca en función de las necesidades.
- Se usa para ello la función `realloc`.

Redimensionar la reserva de memoria

```c
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int * a;
  a = malloc(10 * sizeof (int)); /* Se pide espacio para 10 enteros. */
  /* ... */
  a = realloc(a, 20 * sizeof (int)); /* Ahora se amplía para que quepan 20. */
  /* ... */
  a = realloc(a, 5 * sizeof (int)); /* Y ahora se reduce a sólo 5 (los 5 primeros). */
  /* ... */
  free(a);
  return 0;
}
```